Computing the Well-Founded Semantics Faster

Kenneth A. Berman, John S. Schlipf, and John V. Franco

University of Cincinnati, Department of ECE&CS, USA

Abstract. We address methods of speeding up the calculation of the well-founded semantics for normal propositional logic programs. We first consider two algorithms already reported in the literature and show that these, plus a variation upon them, have much improved worst-case behavior for special cases of input. Then we propose a general algorithm to speed up the calculation for logic programs with at most two positive subgoals per clause, intended to improve the *worst case* performance of the computation. For a logic program $\mathcal{P}$ in atoms $\mathcal{A}$, the speed up over the straight Van Gelder alternating fixed point algorithm (assuming worst-case behavior for both algorithms) is approximately $(|\mathcal{P}|/|\mathcal{A}|)^{(1/3)}$. For $|\mathcal{P}| \geq |\mathcal{A}|^4$, the algorithm runs in time linear in $|\mathcal{P}|$.

1 Introduction

Logic programming researchers have, over the last several years, proposed many logic-based declarative semantics for various sorts of logic programming. The hope is that, if they can be efficiently implemented, these semantics can restore the separation between logic and implementation that motivated the developers of Prolog but that Prolog did not achieve. In the last few years, several projects have begun to produce working implementations of some of these semantics.

A major difficulty with all such approaches is the complexity of the calculations, and various approaches have been tried to speed the calculations up. We feel that serious investigation of new algorithms to compute these semantics may have significant practical importance at this time.

Two quite popular semantics for the class of normal logic program are the stable [4] and the well-founded [14]. For propositional logic programs, computing inferences under the stable semantics is known to be co-NP-complete [7]; computing inferences under the well-founded is known to be quadratic time (folklore), and no faster algorithm is known. Work in [8] (on questions of updating accessibility relations in graphs) suggests that it may be quite difficult, using standard techniques, to break the quadratic-time bound on the well-founded semantics. We are concerned here with speeding up the computation of the well-founded semantics; in particular, we find a large class of propositional logic programs for which we can break the quadratic time bound. As an additional application, we also note that, if the calculation of the well-founded semantics can be made sufficiently fast, that calculation may also prove a highly useful subroutine in computing under the stable semantics.

The standard calculation methods for the well-founded semantics are based upon the alternating fixed-point algorithm of Van Gelder [13]. Several researchers

References

1. J. Dix. A classification theory of semantics of normal logic programs: II. Weak properties. To appear in *JCSS*.
2. W. F. Dowling and J. H. Gallier. Linear time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming* 1 (1984), 267–284.
3. M. Fitting. A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming*, 2(4):295–312, 1985.
4. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. 5th Int'l Conf. Symp. on Logic Programming*, 1988.
5. V. Lifschitz and H. Turner. Splitting a logic program. Preprint.
6. A. Itai and J. Makowsky, On the complexity of Herbrand's theorem. Technical Report No. 243, Department of Computer Science, Israel Institute of Technology, Haifa (1982).
7. W. Marek and M. Truszczyński. Autoepistemic logic. *Journal of the ACM* 38(3), pages 588-619, 1991.
8. J. Reif. A topological approach to dynamic graph connectivity. *Information Processing Letters* 25(1), pages 65-70.
9. J. S. Schlipf. The expressive powers of the logic programming semantics. To appear in *JCSS*. A preliminary version appeared in *Ninth ACM Symposium on Principles of Database Systems*, pages 196-204, 1990. Expanded version available as University of Cincinnati Computer Science Technical Report CIS-TR-90-3.
10. M. G. Scutellà. A note on Dowling and Gallier's top-down algorithm for propositional Horn satisfiability. *Journal of Logic Programming* 8, pages 265-273, 1990.
11. V. S. Subrahmanian, personal communication.
12. R. Tarjan, "Depth first search and linear graph algorithms," *SIAM Journal on Computing* 1 (1972), 146–160.
13. A. Van Gelder. The alternating fixpoint of logic programs with negation. In *Eighth ACM Symposium on Principles of Database Systems*, pages 1–10, 1989. Available from UC Santa Cruz as UCSC-CRL-88-17.
14. A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM* 38(3), pages 620-650, 1991.

This article was processed using the LaTeX macro package with LLNCS style

a propositional logic program is generally not particularly efficient, since uniformities over the variables are lost. Accordingly, the utility of the methods we use here will be determined in part by whether they can be incorporated in faster methods for handling first order logic programs. This we intend to pursue in future research.

2 Terminology and Notation

A normal propositional logic program $\mathcal{P}$ over a set of atoms $\mathcal{A}$ consists of a finite set of *rules* of the form

$$a \leftarrow \beta_1 \wedge \beta_2 \wedge \cdots \wedge \beta_i,$$

where each β_i is a literal, i.e., a proposition letter in $\mathcal{A}$ or its negation. We shall use these meanings of $\mathcal{P}$ and $\mathcal{A}$ henceforth in the paper.

Definition 1. 1. Traditionally associated with a logic program $\mathcal{P}$ is a set of binary dependency relations on the atoms of the programs. For proposition letters a, b:
   - $b <_{pos} a$ (*a depends directly positively* upon b) if there is a rule in $\mathcal{P}$ with head a and subgoal b.
   - $b <_{neg} a$ (*a depends directly negatively* upon b) if there is a rule in $\mathcal{P}$ with head a and subgoal $\neg b$.

2. Two further dependency relations are defined from the above relations, using the obvious regular-set type notation, with sequence meaning concatenation:
   - $<_{dep} = <_{pos} \cup <_{neg}$. Relation $<_{dep}$ is called the *direct dependency relation.*
   - $<_{1neg} = <_{neg}<_{pos}{}^*$. Relation $<_{1neg}$ is the relation of *dependency through exactly one negation* (and any number of positives).
3. A logic program $\mathcal{P}$ is *stratified* if its relation $<_{1neg}$ is acyclic (i.e, if its transitive closure is irreflexive).
4. A logic program $\mathcal{P}$ is *positive-acyclic* if its relation $<_{pos}$ is acyclic.

For both stratified and positive-acyclic logic programs, it turns out that the well-founded semantics can be found in time linear in the size of the logic program.

In this paper we make some fairly standard assumptions about how complexity is measured. We assume that every propositional logic program has proposition letters $x_1, \ldots, x_n$, for some n. We assume that computations are performed on a machine with random-access memory, and, in particular, that accessing array positions, following pointers, and copying pointers, can done in constant time. (Similar assumptions were made, for example, in [2]). Finally, we assume that certain arithmetic operations (initialization to 0, adding, dividing, copying, incrementation, decrementation, and comparison) on natural numbers used for indices and reference counts can be done in unit time.

3 A Hypergraph Representation of Programs

To picture algorithms to compute the well-founded semantics, we frequently think of logic programs as representing directed hypergraphs in a fairly straightforward way:

Definition 2. Let $\mathcal{P}$ by a logic program over atoms set $\mathcal{A}$. With only trivial rewriting, we may assume there is exactly one rule $s \leftarrow$ with no positive subgoals, and for that s, $\neg s$ does not appear in $\mathcal{P}$. (If necessary, add such a proposition letter to a. If any other rule has no positive subgoals, add in s as a positive subgoal.) We shall call s the *source*. We shall ignore rule $s \leftarrow$ in building the hypergraph, instead treating the atom s specially.

Set $\mathcal{A}$ is the set of vertices of the hypergraph. First group together rules that have the same heads and positive subgoals: group rules

$$\begin{array}{l} a \leftarrow b_1 \wedge \cdots \wedge b_i \wedge \neg c_1^1 \wedge \cdots \wedge \neg c_{j_1}^1 \\ a \leftarrow b_1 \wedge \cdots \wedge b_i \wedge \neg c_1^2 \wedge \cdots \wedge \neg c_{j_2}^2 \\ \quad\vdots \\ a \leftarrow b_1 \wedge \cdots \wedge b_i \wedge \neg c_1^h \wedge \cdots \wedge \neg c_{j_h}^h \end{array}$$

into "rules" the form

$$a \leftarrow b_1 \wedge \cdots \wedge b_i \wedge ((\neg c_1^1 \wedge \cdots \neg c_{j_1}^1) \vee (\neg c_1^2 \wedge \cdots \neg c_{j_2}^2) \vee \cdots \vee (\neg c_1^h \wedge \cdots \neg c_{j_h}^h))$$

The head and positive subgoals of the "rule" will be considered to be a *directed hypergraph edge* $\langle a, \{b_1, \ldots, b_i\}\rangle$, directed from the subgoals to the head. The remaining conjunct,

$$((\neg c_1^1 \wedge \cdots \wedge \neg c_{j_1}^1) \vee (\neg c_1^2 \wedge \cdots \wedge \neg c_{j_2}^2) \vee \cdots \vee (\neg c_1^h \wedge \cdots \wedge \neg c_{j_h}^h)),$$

will be called the *presupposition* of the edge. (Intuitively, the rule is applied only if the presupposition is either known to be true or not known to be false, depending upon circumstances.) Variable a will also be called the *head* of the edge, and $b_1, \ldots, b_i$ will be called the *tails* of the edge.

The *degree* $d(a)$ of any $a \in \mathcal{A}$ is the sum of the numbers of tails on all hyperedges with head a.

We shall use both hypergraph-theoretic language and logic-program language in this paper, whichever seems more obvious at any point.

Definition 3. Given a partial function $\tau : \mathcal{A} \rightarrow \{T, F\}$ we can easily assign an interpretation $\tau(p)$ for p the presupposition of any rule:

$$\tau(((\neg c_1^1 \wedge \cdots \wedge \neg c_{j_1}^1) \vee (\neg c_1^2 \wedge \cdots \wedge \neg c_{j_2}^2) \vee \cdots \vee (\neg c_1^h \wedge \cdots \wedge \neg c_{j_h}^h))$$

is true if every conjunct of some disjunct is true in τ, false if some conjunct of every disjunct is false in τ, and undefined otherwise. Moreover, this can be computed by substituting T literals true in τ and F for literals false in τ and doing standard simplifications.

have observed that, for propositional logic programs, the (worst case) time taken by that algorithm to compute the well-founded partial model is quadratic in the size of the program. More specifically, for a propositional logic program $\mathcal{P}$ with atom set $\mathcal{A}$, the complexity is $\mathbf{O}(|\mathcal{A}||\mathcal{P}|)$: Van Gelder's algorithm can make at most $|\mathcal{A}|$ passes before reaching a fixed point (plus another to verify that a fixed point has been reached), and each pass consists of finding the least models of two Horn clause programs derived (essentially via the Gelfond-Lifschitz Transform) from $\mathcal{P}$. Since finding the least model of a Horn clause program is linear in the size of the program [2, 6, 10], this gives the stated complexity.

Some techniques are know for speeding up the calculation on "nice" programs. We investigate two of them and prove that they do achieve optimal speed-up on certain classes of programs. We then turn to improving the worst-case behavior. Our main result is Theorem 10: We show that our Algorithm 6 computes the well-founded partial model of any program $\mathcal{P}$ in proposition letters $\mathcal{A}$ in time $\mathbf{O}(|\mathcal{P}| + |\mathcal{A}|^2 + |\mathcal{P}|^{\frac{2}{3}}|\mathcal{A}|^{\frac{4}{3}})$, so long as $\mathcal{P}$ has at most two positive subgoals per rule. Thus when $\mathcal{P}$ is sufficiently larger than $\mathcal{A}$, this new algorithm will thus have a noticably better worst-case time than Van Gelder's original construction. Thus also, $|\mathcal{A}|^4 \in \mathbf{O}(|\mathcal{P}|)$, our algorithm takes time linear in $|\mathcal{P}|$, which is clearly optimal.

Restrictions

We shall be discussing only (finite) propositional logic programs. Now for many practical purposes, propositional logic programs are not especially interesting by themselves, but semantics for first order logic programs (a.k.a. intensional databases, or IDB's) over (finite) extensional databases (EDB's) are generally defined by translating, via Herbrand expansions, these logic programs and databases into (finite) propositional logic programs.

In our principal algorithm, Algorithm 6, we shall consider only logic programs with at most two positive subgoals per rule. In some circumstances, the restriction to two positive subgoals per rule is no particular hindrance; it is well known that, given any logic program $\mathcal{P}$ in proposition letters $\mathcal{A}$, one can easily construct a logic program $\mathcal{P}^+$ in proposition letters $\mathcal{A}^+ \supseteq \mathcal{A}$ which is a conservative extension of $\mathcal{P}$ under many logic programming semantics, including the well-founded, the stable, and Fitting's Kripke-Kleene (3-valued program completion) semantics [3]. However, in general $\mathcal{A}^+$ will be substantially larger than $\mathcal{A}$, making $|\mathcal{A}^+||\mathcal{P}^+|$ substantially larger than $|\mathcal{A}||\mathcal{P}|$ — and thus possibly negating all benefits of our algorithm.

Nevertheless, we feel Algorithm 6 may prove fairly widely useful. For example, there are very natural conditions on IDB's that guarantee that $c\sqrt[3]{(|\mathcal{A}|/|\mathcal{P}|)} < 1$ for all large enough EDB's, and for which the standard reduction to a program with two positive subgoals per rule, done in the IDB and thus automatically extended to the propositional program, leaves $|\mathcal{A}^+| \in \mathbf{O}(|\mathcal{A}|)$.

Of course, in actual practice, computing the well-founded semantics for a particular IDB over various EDB's by first translating each IDB/EDB pair into

The algorithm embodied in Van Gelder's alternating fixed point definition of the well-founded semantics [13] translates almost immediately into the following algorithm on their corresponding hypergraphs:

Algorithm 1. (Van Gelder) We are given a program $\mathcal{P}$ and its hypergraph interpretation $\mathcal{H}$. We maintain a partial truth assignment τ, consisting of the truth values already inferred, and two approximations to $\mathcal{H}$, an over-approximation $\overline{\mathcal{H}}$ and an under-approximation $\underline{\mathcal{H}}$. Initially, $\tau = \emptyset$ and $\overline{\mathcal{H}} = \mathcal{H}$, and $\underline{\mathcal{H}}$ is the set of hyperedges of $\mathcal{H}$ with empty presuppositions. We maintain two loop invariants: $\underline{\mathcal{H}}$ is the set of hyperedges of $\mathcal{H}$ with presuppositions p where $\tau(p) = T$, and $\overline{\mathcal{H}}$ is the set of hyperedges of $\mathcal{H}$ with presuppositions p where $\tau(p) \neq F$.

```
Repeat until no changes are made during an entire pass:
  1. Van Gelder edge deletion step:
       Find all vertices $a \in \mathcal{A}$ accessible from $s$ in hypergraph $\underline{\mathcal{H}}$.
       For every vertex $a$ accessible in $\underline{\mathcal{H}}$
          set $\tau(a) = T$.
       For every hyperedge $e$ of $\overline{\mathcal{H}} - \underline{\mathcal{H}}$ with presupposition $p$ containing $\neg a$,
          if $\tau(p) = F$
             remove $h$ from $\overline{\mathcal{H}}$.
  2. Van Gelder edge addition step:
       Find all vertices $a \in \mathcal{A}$ accessible from $s$ in hypergraph $\overline{\mathcal{H}}$.
       For every vertex $a$ not accessible in $\overline{\mathcal{H}}$
          set $\tau(a) = F$.
       For every hyperedge $e$ of $\overline{\mathcal{H}} - \underline{\mathcal{H}}$ with presupposition $p$ containing $\neg a$,
          if $\tau(p) = T$
             add $h$ to $\underline{\mathcal{H}}$.
```

The well-founded partial model of $\mathcal{P}$ is the final partial interpretation τ.

4 Elaborating Upon Known Speed-Ups

"Best-case" and informal "Average-case" speedup techniques for the well-founded semantics have been fairly widely observed.[1] The algorithmic methods of this section are already known; what is new here is (1) a new algorithm, which explicitly combines two older ideas, and (2) some propositions about these algorithms.

The first speedup we first heard from Subrahmanian [11]. Fitting's Kripke-Kleene semantics [3] (a.k.a. 3-valued program completion semantics) for logic programs can be calculated much as is the well-founded semantics, but with Van Gelder's search for inaccessible nodes of $\overline{\mathcal{H}}$ replaced by searches for nodes (other than the source s) with in-degree 0. (Essentially this algorithm is in [3].)

[1] We say "informal Average-case" because there is no accepted distribution on which to base the average and because, so far as we know, the conclusions are based only upon experimentation.

Proposition 4. (folklore) *The above-sketched computation of the Kripke-Kleene semantics for a propositional logic program $\mathcal{P}$ can be done in $O|\mathcal{P}|$ time.*

Idea of Proof. The proof depends upon the fact that finding whether a node is inaccessible from s takes a search, while a node with in-degree 0 can be identified in constant time with the right data structure. ∎

Essentially the same proof gives that

Proposition 5. *Suppose an algorithm is of the following form, and a logic program $\mathcal{P}$ is given:*

```
Set $\tau = \emptyset$
Repeat until a fixed point is reached
    Use Fitting's algorithm to extend $\tau$
    Set $\tau(a) = F$ for some proposition letters $a$ previously not in the domain($\tau$).
```

Then the total *time taken by* all *iterations of the Fitting algorithm is* $O(|\mathcal{P}|)$.[2]

Algorithm 2. (Subrahmanian et. al.)

```
Initialize $\tau = \emptyset$, $\overline{\mathcal{H}} = \mathcal{H}$, and $\underline{\mathcal{H}} =$ the hyperedges of $\mathcal{H}$ with empty
        presuppositions.
Repeat until no edges are deleted in step 2:
  1. Fitting Algorithm:
        Repeat until no changes are made during an entire pass:
            For every vertex $a$ where $\tau(a)$ is undefined and for some hyperedge
                    $h$ of $\underline{\mathcal{H}}$, for all tails $b$ of $h$, $\tau(b) = T$,
                set $\tau(a) = T$.
            For every vertex $a$ where $\tau(a)$ is undefined and $d(a) = 0$ in $\overline{\mathcal{H}}$
                set $\tau(a) = F$.
            For every hyperedge $e$ of $\overline{\mathcal{H}}$ with presupposition $p$,
                if $\tau(p) = F$
                    remove $h$ from $\overline{\mathcal{H}}$.
            For every hyperedge $e$ of $\overline{\mathcal{H}} - \underline{\mathcal{H}}$ with presupposition $p$,
                if $\tau(p) = T$
                    add $h$ to $\underline{\mathcal{H}}$.
  2. Make one pass of Van Gelder edge addition step
```

The well-founded partial model of $\mathcal{P}$ is the final partial interpretation τ.

Subrahmanian noted that Algorithm 2 is faster than Algorithm 1 on most of the logic programs his group randomly generated. We find a partial explanation for this in the following proposition:

[2] We have sloughed over an important detail here since it does not affect this paper: we may have to use a four-valued logic, with the four truth values T, F, undefined, and contradictory.

Proposition 6. *Let $\mathcal{P}$ be a logic program.*

1. *If $\mathcal{P}$ is positive-acyclic, then the Kripke-Kleene semantics for $\mathcal{P}$ agrees with the well-founded semantics, and hence Algorithm 2 finds the well-founded semantics in time $\mathbf{O}(|\mathcal{P}|)$.*
2. *If $\mathcal{P}$ consists of a positive-acyclic program plus some additional rules giving only k hyperedges (possibly with very long presuppositions), then Algorithm 2 finds the well-founded semantics in time $\mathbf{O}(k|\mathcal{P}|)$.*

Proof. 1. In [9] we showed that, for a logic program with no positive subgoals, the Kripke-Kleene and well-founded semantics agree. It is routine to extend that proof to all positive-acyclic logic programs.

2. The difference between the Kripke-Kleene semantics and the well-founded semantics is that the well-founded semantics identifies positive dependency cycles. Since the first part of each pass through the outside loop of the algorithm is Fitting's algorithm, and since $\mathcal{A}$ is finite, each atom found by step 2 must be in a positive cycle, all of whose atoms are inaccessible. Thus one of the k special edges (the "back-edges") must be in this cycle. Accordingly, at least one back-edge will be added to $\overline{\mathcal{H}}$ in each pass. Thus, after k passes, no additional cycles in $\overline{\mathcal{H}}$ remain to be discovered, and all additional inferences may be made by the Fitting semantics. ∎

A second speed-up is motivated by the fact that calculating the perfect model of a stratified logic program (which is also the well-founded partial model and the unique stable model) can be done in linear time. Consider $\langle A, <_{dep}\rangle$ as a directed graph. In linear time, form the strongly connected components of the graph as in [12], and topologically sort the components. Let $\mathcal{P}_1$ be the set of rules of $\mathcal{P}$ whose heads occur in the first component of the graph. $\mathcal{P}_1$ turns out to be pure Horn, so its perfect model is its minimal model M_1, which can be computed in time linear in $|\mathcal{P}_1|$ [2, 6, 10]. Let $\mathcal{P}_2$ be the set of rules of $\mathcal{P}$ whose heads occur in the second component. Substitute into $\mathcal{P}_2$ the truth values computed for variables in the first component and simplify. The result is a pure Horn program, so compute its minimal model, in time linear in $|\mathcal{P}_2|$, and the perfect model of $\mathcal{P}_1 \cup \mathcal{P}_2$ is $M_1 \cup M_2$. Continue this way through all the strongly connected components. The result is the perfect model, which has been constructed in time linear in $|\mathcal{P}|$.

Now for the well-founded and Kripke-Kleene semantics, the same partitioning into strongly-connected components, decomposition of the program by the components and construction of the semantics for each piece separately correctly constructs the semantics. We shall call the levels $\mathcal{P}_1, \mathcal{P}_2, \ldots$ *strata*, or *modules.* (These were investigated in the current context in [9] and [1].) Unlike with stratified programs, there is no guarantee that the strata are Horn. However, for programs with more than one strongly connected component, this gives a divide-and-conquer approach which can sometimes, as with stratified programs, reduce the complexity of finding the well-founded model. (The same technique has been used [5] to speed up the search for individual stable models, though the technique does not work directly for computing the intersection of stable models [9].)

We can combine this technique with Algorithm 2, giving what appears to be a fairly useful algorithm. We have chosen to start first with the Fitting step since, for example, calculations from non-stratified IDB's over nice EDB's can reduce to stratified programs after only one or two passes of the Fitting method, e.g., for modularly stratified programs.

In the following algorithm, we refer to the direct dependency relation $<_{dep}$ on $\overline{\mathcal{H}}$. This is the analogue of the logic programming definition: $a <_{dep} b$ if there is a hyperedge in $\overline{\mathcal{H}}$ with head a and with either b in its tail or $\neg b$ appearing in the presupposition for the hyperedge. Since the algorithm is merely a combination of previous pieces, we summarize the algorithm. The algorithm is initially called with $\tau = \emptyset$ and $\overline{\mathcal{H}}$ and $\underline{\mathcal{H}}$ as in Algorithms 1 and 2. The details of partitioning into strata are can be filled in easily by the reader.

Algorithm 3.

```
Initialize $\tau = \emptyset$.
Perform the Fitting step, updating $\tau$ and *all* current subgraphs of $\underline{\mathcal{H}}$ and $\overline{\mathcal{H}}$.
Decompose $\mathcal{A}$ into its strongly connected components $\mathcal{A}_1, \ldots, \mathcal{A}_j$ under $<_{dep}$
          (defined from $\overline{\mathcal{H}}$).
Topologically sort the components, $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n$.
Construct the subprograms $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n$.
For $i = 1$ to $n$:
  Construct the subhypergraphs $\underline{\mathcal{H}}_i$ and $\overline{\mathcal{H}}_i$.
  Perform one Van Gelder edge addition step on $\mathcal{P}_i$,
        updating $\tau$ and *all* current subgraphs of $\underline{\mathcal{H}}$.
  If any changes were made by the Van Gelder step
    Call Algorithm 3 recursively on $\mathcal{P}_i$.
```

The well-founded partial model of $\mathcal{P}$ is the final partial interpretation τ.

Proposition 7. *Consider the class of programs $\mathcal{P}$ consisting of strata $\mathcal{P}_1, \ldots, \mathcal{P}_i$ where each stratum is either Horn or positive-acyclic. Algorithm 3 finds the well-founded partial model of any such $\mathcal{P}$ in time* $\mathrm{O}(|\mathcal{P}|)$.

5 Improving the Quadratic Bound on the Worst Case

The techniques of the previous section often speed up logic program evaluation, but they do not improve on the worst-case behavior of Van Gelder's algorithm. Moreover, writing an IDB for testing some property P of interest which, for all interesting EDB's, the properties of Proposition 7 hold, may be very difficult for some properties P. We would like a far more general speedup technique.

Van Gelder's algorithm repeatedly deletes edges from $\overline{\mathcal{H}}$ and rechecks for accessibility from s. Reif [8] has studied algorithms for updating vertex accessibility information in directed graphs during dynamic edge deletion. His work suggests it may in general be very difficult to speed up in the worst case past the size of the graph times the number of cycles of deletions. Thus his result seems to suggest that generally fast well-founded semantics algorithms might be difficult or impossible to find, even for logic programs with *at most one* positive subgoal per rule. This same lower bound applies to all algorithms for the well-founded semantics which we have previously seen, arousing our interest in the question of whether that quadratic time bound can be beaten. In what follows, we do improve in this bound, albeit, as previously noted, only for logic programs with at most two positive subgoals per rule and where $c\sqrt[3]{(|\mathcal{A}|/|\mathcal{P}|)} < 1$ for some constant c. The heart of the difference between Reif's work and ours is that we are looking for a fixed point of an operator and may stop as soon as one is reached.

¿From now on we limit attention to the class of propositional logic programs $\mathcal{P}$ over atoms $\mathcal{A}$ with at most 2 positive subgoals per clause.

Observe that, with Van Gelder's algorithm, it takes a relatively large amount of work to find an unfounded set: an entire depth-first search of the hypergraph $\overline{\mathcal{H}}$. If the unfounded set is large, there is a fairly good payoff for the work of searching, but if the unfounded set is small, there has been a high cost to identify a few truth values. It is fairly easy to construct examples where all the algorithms of the previous section take $|\mathcal{A}|$ passes to discover unfounded sets, each of small size. Our goal here is to find somewhat faster ways to find small unfounded sets.

The basic idea is this: Proceed as in Algorithms 2 and 3. Usually, before doing the Van Gelder edge addition step or attempting to decompose the program into strata (modules) (the two time-consuming steps), first do a depth-first search of a "small" approximation $\overline{\mathcal{H}'}$ to $\overline{\mathcal{H}}$, an approximation chosen so that any atom inaccessible in $\overline{\mathcal{H}'}$ is also inaccessible in $\overline{\mathcal{H}}$; since $\overline{\mathcal{H}'}$ is smaller than $\overline{\mathcal{H}}$, the depth-first search will proceed faster. If any inaccessible nodes are found, adjust τ and hypergraph $\underline{\mathcal{H}}$ as before and go back to repeat the Fitting computation, having found inaccessible nodes more quickly than the with the other algorithms. In general, only if no inaccessible nodes are found do we do a full depth-first search. If no inaccessible nodes are found by the full depth-first search either, then the algorithm is finished, with just an additional $\mathrm{O}(|\mathcal{P}|)$ steps used in checking that we're done. If, on the other hand, some inaccessible nodes are found, i.e., a non-empty unfounded set is found, we shall show that a "large" number of inaccessible nodes must be found. Thus the cost of the depth-first search, averaged over all the inaccessible nodes found, is not too high.

If e_1, e_2 are hypergraph edges with the same head, and if every tail of e_1 is also a tail of e_2, then say that e_1 is a *subedge* of e_2, and e_2 *extends* e_1.

We do the approximation to hypergraph $\overline{\mathcal{H}}$ described above in two steps. We start with a subalgorithm that applies only to hypergraphs H where each subedge $\langle a, \{b\}\rangle$ occurs in $\leq \mu$ edges of H for some μ. It will use a parameter T to be optimized later.

Algorithm 4. Let $X = \{a \in \mathcal{A} : a \text{ is the head of } \geq T \text{ edges of } \mathcal{H}\}$.
Let $\mathcal{H}'' = \{\langle a, \{s\}\rangle : a \in X\} \cup \{e \in H : \text{ head}(e) \notin X\}$.

```
Do a depth-first search of $\mathcal{H}''$ to find all inaccessible nodes.
If no inaccessible nodes are found,
    perform a depth-first search of $\mathcal{H}$ to find inaccessible nodes.
```

Lemma 8. *1. If a vertex is inaccessible in $\mathcal{H}''$, it is inaccessible in $\mathcal{H}$.*

2. *If an inaccessible node is found in the first search, then the total cost of searching is $\mathbf{O}(T|\mathcal{A}|)$.*
3. *The total cost of searching, if the second search is called, is $\mathbf{O}(|\mathcal{H}|)$.*
4. *If no inaccessible node is found in the first search but an inaccessible node is found in the second one, then $\geq \frac{T}{\mu}$ inaccessible nodes are found. Thus the total cost of searching* per inaccessible node found *is $\mathbf{O}(\frac{\mu|\mathcal{H}|}{T})$.*

Proof. 1. Obvious.

2. $|\mathcal{H}''| \in \mathbf{O}(T|\mathcal{A}|)$ since there are at most T edges in $\mathcal{H}''$ into each node.
3. Suppose a is inaccessible in $\mathcal{H}''$ but accessible in $\mathcal{H}$. Then $a \in X$, so there are $\geq T$ edges with head a. Thus at least one tail on each of these edges must be inaccessible in $\mathcal{H}$, since otherwise a would be accessible. Suppose $< \frac{T}{\mu}$ inaccessible nodes are found. Then, by the pigeon-hole principle, one of these inaccessible nodes must occur on $> T/(\frac{T}{\mu}) = \mu$ edges, contradicting the assumption on μ.
4. Immediate from the previous part since the cost of one depth-first search is $\mathbf{O}(|\mathcal{H}|)$. ∎

To get a near-optimal value for T above, equate $T|\mathcal{A}|$ and $(\mu|\mathcal{H}|)/T$. Solving for T we get $T^2 = \mu|\mathcal{H}|/|\mathcal{A}|$. This gives an average cost of searching (per inaccessible vertex found) of $\sqrt{\mu|\mathcal{H}||\mathcal{A}|}$.

If in the original hypergraph $\mathcal{H}$ the value of μ (defined just before Algorithm 4) is small, the total cost of the algorithm is good. But there is, in general, no reason to expect that value of μ to be small. So we perform another approximation.

Algorithm 5. Let $\mathcal{H}$ be a hypergraph, and let B be a number. Let

$$\begin{aligned}
\mathcal{H}_1 &= \{\langle a, \{b_1\}\rangle : \text{ for } \geq B \text{ vertices } b_2, \langle a, \{b_1, b_2\}\rangle \in \mathcal{H}\} \\
\mathcal{H}_2 &= \{e \in \mathcal{H} : \text{ no subedge of } e \text{ is in } \mathcal{H}_1\} \\
\mathcal{H}' &= \mathcal{H}_1 \cup \mathcal{H}_2
\end{aligned}$$

```
Call Sublgorithm 4 on $\mathcal{H}'$ with $T = \sqrt{B|\mathcal{H}|/|\mathcal{A}|}$.
If no inaccessible nodes are found above,
    perform a depth-first search of $\mathcal{H}$.
```

Lemma 9. *Let $\mathcal{H}$, B, and $\mathcal{H}'$ be as in Sublgorithm 5.*

1. *If some vertex $a \in \mathcal{A}$ is inaccessible from s in $\mathcal{H}'$, it is inaccessible from s in $\mathcal{H}$.*
2. *No edge $\langle a, \{b\}\rangle$ is a subedge of $\geq B$ edges of $\mathcal{H}'$ (i.e, the property of Algorithm 4 holds with $\mu = B$).*
3. *If every vertex $a \in \mathcal{A}$ is accessible from s in $\mathcal{H}'$, but some vertex $a \in \mathcal{A}$ is inaccessible from s in $\mathcal{H}$, then $\geq B$ vertices of $\mathcal{A}$ are inaccessible in $\mathcal{H}$.*
4. *$|\mathcal{H}'| \leq |\mathcal{H}|$. Thus: (i) The cost to perform the search in this algorithm is in $\mathbf{O}(T|\mathcal{A}|)$ if inaccessible nodes are found by the first step of Algorithm 4. (ii) If no inaccessible nodes are found by the first step of Algorithm 4 but some inaccessible nodes are found by the second step, the cost to perform the search is in $\mathbf{O}(|\mathcal{H}'|) \subseteq \mathbf{O}(|\mathcal{H}|)$. (iii) If no inaccessible nodes are found by Algorithm 4, the total cost to perform the search is $\mathbf{O}(|\mathcal{H}|)$.*
5. *If this subalgorithm performs the search of $\mathcal{H}$ and inaccessible nodes are found, then the average cost of searching for each inaccessible node found is $\mathbf{O}(|\mathcal{H}|/B)$.*

Proof.

1. Obvious.
2. The subedges in $\mathcal{H}_1$ have no tails in common, and no edge in $\mathcal{H}_1$ is a subedge of any edge in $\mathcal{H}_2$. If any B edges in $\mathcal{H}_2$ had a subedge in common, that subedge would have been put into $\mathcal{H}_1$.
3. Suppose a is accessible in $\mathcal{H}'$ but not in $\mathcal{H}$. Then there must be an edge $\langle a, \{b\}\rangle \in \mathcal{H}' - \mathcal{H}$ where b is accessible in $\mathcal{H}$ but a is not. By definition of $\mathcal{H}'$, there must be $\geq B$ edges $\langle a, \{b, b_2\}\rangle \in \mathcal{H}$. Since a is inaccessible in $\mathcal{H}$, none of these $\geq B$ b_2's can be accessible in $\mathcal{H}$ either.
4. That $|\mathcal{H}'| \leq |\mathcal{H}|$ is obvious. Statements (i) and (ii) then follow from Lemma 8. Statement (iii) follows from the fact that the searching cost is dominated by the cost of the search of $\mathcal{H}'$ followed by the search of $\mathcal{H}$.
5. Follows from the two previous steps. ∎

A near-optimal value for B can be computed by equating the two previous average costs, $\sqrt{B|\mathcal{H}||\mathcal{A}|}$ and $\frac{|\mathcal{H}|}{B}$. This gives $B = \sqrt[3]{|\mathcal{H}|/|\mathcal{A}|}$, yielding a final average cost of $\mathbf{O}(|\mathcal{A}|^{(1/3)}|\mathcal{P}|^{(2/3)})$. The total search cost must be at most $|\mathcal{A}|$ times this — thus $\mathbf{O}(|\mathcal{A}|^{(4/3)}|\mathcal{P}|^{(2/3)})$. The speedup is on the order of $\sqrt[3]{|\mathcal{P}|/|\mathcal{A}|}$, as stated earlier.

We put our modifications together into the following algorithm. In order to simplify the bookkeeping, we divide the program into strata only once, after the initial application of Fitting's algorithm.

Algorithm 6.

```
Initialize $\tau = \emptyset$. Initialize $\underline{\mathcal{H}}$ and $\overline{\mathcal{H}}$ as usual.
Perform the Fitting step, updating $\tau, \underline{\mathcal{H}}$ and $\overline{\mathcal{H}}$.
Decompose $\mathcal{A}$ into its strongly connected components $\mathcal{A}_1, \ldots, \mathcal{A}_j$ under $<_{dep}$
          (defined from $\overline{\mathcal{H}}$).
Topologically sort the components, $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n$.
Construct the subprograms $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n$.
For $i = 1$ to $n$:
  Construct the subhypergraphs $\underline{\mathcal{H}_i}, \overline{\mathcal{H}}_i$ and construct $\overline{\mathcal{H}}_i'$ (as in Algorithm 5).
  Repeat until no inaccessible edges are found in the next step:
    Call Algorithm 5 on $\overline{\mathcal{H}}_i$, $\overline{\mathcal{H}}_i'$
    If any inaccessible vertices are found,
      Update $\tau$ and all $\underline{\mathcal{H}_j}$'s, $\overline{\mathcal{H}}_j$'s, and $\overline{\mathcal{H}}_j'$'s for $j \geq i$.
      Perform the Fitting step, updating $\tau$ and all $\underline{\mathcal{H}_j}$'s, $\overline{\mathcal{H}}_j$'s, $\overline{\mathcal{H}}_j'$'s for $j \geq i$.
```

The well-founded partial model of $\mathcal{P}$ is the final partial interpretation τ.

It turns out that all the bookkeeping and graph construction and update for the above algorithm can be done in time $|\mathcal{P}| + |\mathcal{A}|^2$, as we shall comment upon in the next section. Combining this with the preceding remarks, we have:

Theorem 10. *Algorithm 6 computes the well-founded semantics for any propositional logic program $|\mathcal{P}|$ in proposition letters $\mathcal{A}$ with ≤ 2 positive subgoals per clause in time* $\mathbf{O}(|\mathcal{P}| + |A|^2 + |A|^{\frac{4}{3}}|\mathcal{P}|^{\frac{2}{3}})$.

Since we expect that in common examples $|\mathcal{A}|^{(4/3)}|\mathcal{P}|^{(2/3)} > |\mathcal{P}| + |\mathcal{A}|^2$, we have simplified the result above to saying that generally our speed-up over the straight Van Gelder algorithm is $\mathbf{O}(|\mathcal{P}|^{(1/3)}|\mathcal{A}|^{-(1/3)})$. Obviously, if $|\mathcal{A}|^4 \in \mathbf{O}(|\mathcal{P}|)$, the algorithm performs in time $\mathbf{O}(\mathcal{P})$, which is best possible.

In practice, it would seem wise to optimize the algorithm by, approximately once every $\sqrt[3]{|\mathcal{P}|/|\mathcal{A}|}$ iterations of the outer repeat loop, replacing the call to Algorithm 5 with a full depth-first search of $\overline{\mathcal{H}}_i$. Except for $\mathcal{P}$ much larger than $\mathcal{A}$, this would not hurt the worst-case performance of the algorithm, and it might help if the algorithm is finding many small unfounded sets when finding some large unfounded set would substantially reduce the size of $\overline{\mathcal{H}}$.

Implementation

What we have left to do is to show that all the necessary bookkeeping for the algorithm can be done in time $\mathbf{O}(|\mathcal{P}|)$. The basic work is in choosing the appropriate data structures and the right counts to store and in doing enough work in preprocessing, leaving mostly updating of the data structures and the counts for run time (plus, of course, many depth-first searches). All this can be done; we just sketch the construction below.

For each $a \in \mathcal{A}$ we store 3 doubly-linked lists of pointers: pointers to hyperedges with head a, hyperedges with a in the tail, and hyperedges with $\neg a$ in the presupposition. (In fact, we have to store the presuppositions themselves as doubly linked lists, simplifying according to the usual logic rules each time an atom is first set to T of F.) We store each $\underline{\mathcal{H}}_i$, $\overline{\mathcal{H}}_i$, and $\overline{\mathcal{H}}_i'$ and update them dynamically; thus we actually need one set of the 3-doubly linked lists of pointers above for each of the three graphs. For this we need a pointer from each edge in $\overline{\mathcal{H}}_i$ to its subedge (if any) in $\overline{\mathcal{H}}_i'$, a counter on the subedge telling how many edges point to it (so we can tell when to remove the subedge from $\overline{\mathcal{H}}'$ and reinstall the original edges), and doubly-linked lists going from the subedge the edges from which it is derived to allow us to reinstall the original edges quickly. (Importantly, each reinstallation is done only once.) We store and dynamically update an $|\mathcal{A}|$-length array giving the number of rules of $\overline{\mathcal{H}}_i'$ with head each variable a; this allows us to create the approximations $\overline{\mathcal{H}}_i''$ of Algorithm 4 quickly. In the construction we use an $|\mathcal{A}| \times |\mathcal{A}|$ array giving the number of times each atom appears as a positive subgoal in a rule with head each $a' \in \mathcal{A}$.

6 Future Work

There are two clear open directions left by this work, both of which we intend to pursue. One is to relax the restriction to two positive subgoals per rule. We expect this can be done, but it is not clear how high the cost is going to be; we do not expect to be able to achieve the same speedup we have here. (By analogy, a better speedup can be achieved if we restrict attention to programs with only one positive literal per clause.) The other direction, as noted before, is to integrate an algorithm such as ours with actual deductive-database inference techniques.

The authors thank Fred Annexstein and R.P. Saminathan for helpful suggestions on this work.

The research of the second and third authors on this paper was partially supported by grant grant N00014-94-1-0382 from the Office of Naval Research.

Authors' address: Department of Electrical and Computer Engineering and Computer Science, The University of Cincinnati, Cincinnati, OH 45221-0008, USA.

References

1. J. Dix. A classification theory of semantics of normal logic programs: II. Weak properties. To appear in *JCSS*.
2. W. F. Dowling and J. H. Gallier. Linear time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming* 1 (1984), 267–284.
3. M. Fitting. A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming*, 2(4):295–312, 1985.
4. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. 5th Int'l Conf. Symp. on Logic Programming*, 1988.
5. V. Lifschitz and H. Turner. Splitting a logic program. Preprint.
6. A. Itai and J. Makowsky, On the complexity of Herbrand's theorem. Technical Report No. 243, Department of Computer Science, Israel Institute of Technology, Haifa (1982).
7. W. Marek and M. Truszczyński. Autoepistemic logic. *Journal of the ACM* 38(3), pages 588-619, 1991.
8. J. Reif. A topological approach to dynamic graph connectivity. *Information Processing Letters* 25(1), pages 65-70.
9. J. S. Schlipf. The expressive powers of the logic programming semantics. To appear in *JCSS*. A preliminary version appeared in *Ninth ACM Symposium on Principles of Database Systems*, pages 196-204, 1990. Expanded version available as University of Cincinnati Computer Science Technical Report CIS-TR-90-3.
10. M. G. Scutellà. A note on Dowling and Gallier's top-down algorithm for propositional Horn satisfiability. *Journal of Logic Programming* 8, pages 265-273, 1990.
11. V. S. Subrahmanian, personal communication.
12. R. Tarjan, "Depth first search and linear graph algorithms," *SIAM Journal on Computing* 1 (1972), 146–160.
13. A. Van Gelder. The alternating fixpoint of logic programs with negation. In *Eighth ACM Symposium on Principles of Database Systems*, pages 1–10, 1989. Available from UC Santa Cruz as UCSC-CRL-88-17.
14. A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM* 38(3), pages 620-650, 1991.

This article was processed using the LaTeX macro package with LLNCS style